Available online at http://scik.org
Commun. Math. Biol. Neurosci. 2025, 2025:124
https://doi.org/10.28919/cmbn/9433
ISSN: 2052-2541

ACTIVATION FUNCTIONS FOR DEEP NEURAL NETWORKS

SAFAA ECHAMSI*, AZIZA EL BAKALI KASSIMI, ESSADIK BELOUAFI, ABDELHAKIM ALALI, ABDELAZIZ BOUROUMI, ASMAE GUENNOUN

Information Processing Laboratory, Faculty of Sciences Ben M'sick, Hassan II University, Casablanca, Morocco

Abstract. Deep Neural Networks are connectionist models composed of multiple layers of neuron-like computational units that try to simulate two fundamental aspects of human intelligence: i) learning from examples, and ii) generalizing the learned knowledge and skills to new and unseen examples. The design of such a model for any real-world application requires several steps, including the choice of an adequate architecture in terms of the number of layers, as well as the size, the type, and the activation function of each layer. This paper investigates the impact of activation functions on the overall performance of the network, which has remained underestimated until the early 2010s. The paper provides a brief historical review of recent literature, along with some practical recommendations for selecting the most appropriate activation functions according to each situation. To illustrate the studied problem, we also present some experimental results showing that a shallow neural network with appropriate activation functions may be more efficient than a deeper neural network using traditional and inappropriate functions.

Keywords: machine learning; artificial neural networks; deep learning; hyperparameters; activation functions.
2020 AMS Subject Classification: 68T05, 68Q32.

1. INTRODUCTION

Artificial Neural Networks (ANNs) are machine learning (ML) models that simulate the learning process in humans, which is one of the main aspects of human intelligence. In supervised learning mode, given a set of m training data pairs in the form:

*Corresponding author
E-mail address: safaa.echamsi-etu@etu.univh2c.ma
Received June 19, 2025

$$X = \left\{ \left(x^{(1)}, y^{(1)}\right), \left(x^{(2)}, y^{(2)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right) \right\},$$

where each $x^{(i)}$ represents an input data and $y^{(i)}$ the corresponding output, the goal of an ANN is to find the best possible estimation of a hypothetical function that maps inputs to outputs as exemplified by the m samples of X [1]. The inputs are generally presented to the network in the form of n-dimensional arrays or tensors, while the outputs are vectors whose components are either real numbers for regression problems, or integers in the case of classification problems (Fig. 1).

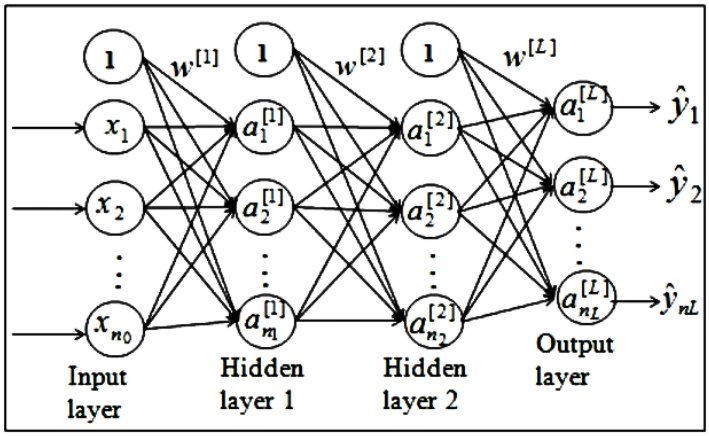

FIGURE 1. Example of a simple multi-layer neural network architecture, composed of $L = 4$ fully connected layers, including an input layer, two hidden layers, and an output layer. The input layer receives the input data in the form of $(n+1)$–dimensional vectors and forwards them to the first hidden layer without transformation. Each of the n_l neurons of each hidden layer l operates a transformation on its input vector $a[l-1]$ and transmits it to all neurons of the following layer using an activation function $a[l]_i$. The last layer, L, produces the network output in the form of a vector of n_L components $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{n_L}$. These components, whose number and nature are problem dependent, are used in the evaluation of the network error, which is exploited by the learning algorithm for the adjustment of the connection weights and biases of all processing units. For the sake of simplifying the notations, the input vector of each processing layer contains an additional component, whose value is fixed to 1. The connection weight of this particular component to each neuron of the subsequent layer represents the neuron's bias.

For both problems, ANNs are considered as universal approximators in the sense that they are theoretically able to find an approximate yet satisfying estimation for any hypothetical function,

regardless of its form or complexity[2]. In practice, however, the design of a neural network-based solution for any real-world problem consists of several steps that require careful engineering and both intuition and experience, especially for deep neural networks composed of multiple processing layers of different types, with many hyperparameters that need to be set by the designer for each layer[3].

Depending on the problem at hand and the available training data, the first step concerns the choice of an adequate architecture for the network, i.e., the number and nature of neurons to use and the way to connect them in order to form the network[4].

The second step is the choice of an objective function for measuring the network errors during its training process, i.e., the differences between the predicted outputs $\hat{y}^{(i)}$ and the desired outputs $y^{(i)}$ for $1 \leq i \leq m$. Generally called "loss function", the objective function represents a criterion that should be minimised using an optimisation algorithm, or optimiser. The role of the optimizer is to adjust the trainable parameters of the network (synaptic weights and biases) according to an efficient learning rule that takes into account the contribution of each parameter to the overall error observed at the output of the network.

Finally, one or more metrics should be selected for the purpose of evaluating the performance of the network on unseen data in order to assess its ability to generalise the learned knowledge and skills to new situations[5].

This paper aims to contribute to the research efforts dedicated to preventing and mitigating the impacts of hyper-parameters on the overall performance of deep neural networks[6]. More specifically, the paper focuses on the proper choice of the activation function for each computing layer. Activation functions are simple yet important hyper-parameters that have been rather neglected for many years, despite the significant influence they could have on both the convergence time and the results quality of the training process[7].

In the next section we provide a historical review of recent literature in the area of activation functions; with a brief description of the most currently used functions and some recommendations on how to select the most appropriate function for each processing layer depending on the context of each application.

In section 3 we present and discuss some experimental results, highlighting the fact that the performance of a shallow neural network with only one or two hidden layers may exceed that of a deeper neural network if the activation functions are more appropriately selected for the shallow architecture. Finally, in section 4 we conclude the paper and provide some ideas for further research.

2. HISTORICAL REVIEW

The concept of activation function was first introduced by McCulloch and Pitts in 1943 in their seminal work on the biological neuron model. Due to the "all-or-none" character of nervous activity, McCulloch and Pitts based their model on the idea that: "At any instant a neuron has some threshold, which excitation must exceed to initiate an impulse"[8]. The resulting model is depicted in figure2.

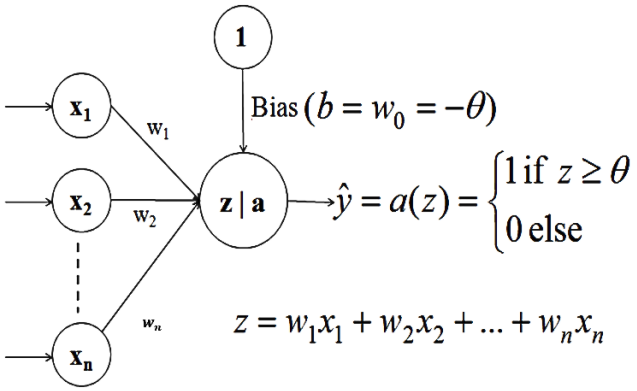

FIGURE 2. Threshold linear unit, or perceptron, composed of a single processing neuron with n weighted inputs, a bias b, a threshold θ and a thresholded activation function a. Note that by posing $b = w_0 = -\theta$ the inequality $z \geq \theta$ can be rewritten as $w_1x_1 + w_2x_2 + \cdots + w_nx_n + b \geq 0$ or $w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n = w^T x \geq 0$

It can be seen as a threshold logic unit (TLU) in the sense that the estimated output, $\hat{y}$, is equal either to 1 or to 0 depending on the linear sum of the weighted inputs. However, it is only 15 years later, in 1958, that Rosenblatt proposed the first learning algorithm that allowed this model to be used as a binary classifier, called "Perceptron". This algorithm provided the first solution

to the problem of automatic determination of the trainable parameters of the model (weights and bias) from a set of training data[9]. But due to their limitations as linear classifiers that could only be trained to separate between linearly separable classes, Rosenblatt's perceptrons were severely criticized by Minsky and Papert in 1969 [10].

Although it was known that non-linear classification problems could be solved using multi-layer perceptrons (MLP), no learning algorithm was available for training such a model. Moreover, there was intuitive judgement that the extension to MLP was sterile[11], which has led to the so-called "First AI Winter" that lasted several years.

In 1986 this limitation was overcome with the publication of the now well-known and ubiquitous back-propagation algorithm (BP). BP is an optimization procedure aimed at minimizing the following objective function that measures the global error of a MLP (Fig.1) among the m samples of its training dataset X:

$$E(X,w) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n_L}\left(\hat{y}_j^{(i)} - y_j^{(i)}\right)^2 \tag{1}$$

where w denotes the set of all weights and biases, $\hat{y}^{(i)}$ is the computed output for the ith training example, and $y^{(i)}$ its desired output.

The minimization of E can be performed using the gradient descent optimiser, according to which at each epoch t each weight j of each neuron i of each layer l is adjusted according to the learning rule

$$w_{ij}^{[l]}(t) = w_{ij}^{[l]}(t-1) - \alpha\frac{\partial E}{\partial w_{ij}^{[l]}(t-1)} \tag{2}$$

where $\alpha \in [0,1]$ represents the learning rate.

Using back-propagation algorithm, the implementation of this rule requires the computation of the partial derivatives of E with respect to the activation function $a_j^{[l]}$ of each unit of each layer j, for $j = L, L-1, L-2, \ldots, 1$. For this, these activation functions should be derivable and non-linear[12]. In, the same activation function was used for all processing units of the network. It is the well-known logistic or sigmoid function defined by

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} \tag{3}$$

whose derivative is

$$\frac{d\sigma(z)}{dz} = \frac{-(-e^{-z})}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \times \frac{(e^{-z}+1)-1}{1+e^{-z}} = \sigma(z)(1-\sigma(z)) \tag{4}$$

The graphs of this function and its derivative are depicted in fig 3.

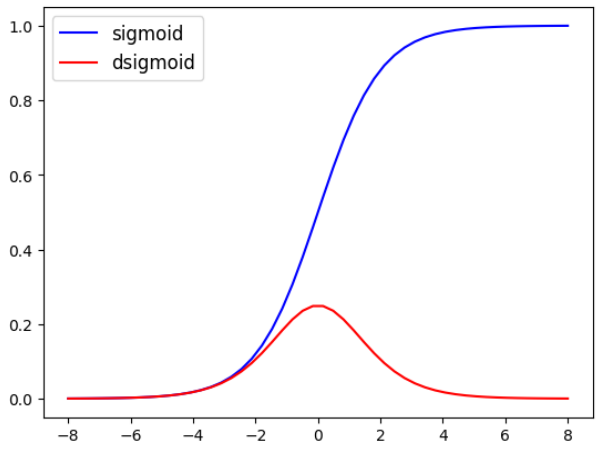

FIGURE 3. The sigmoid activation function and its derivative.

A striking property shown by this figure is that the derivative values are limited to the interval $]0, 0.25[$, which can be an issue in certain situations as we will see in the next section.

Another property is the limitation of the outputs of $\sigma(z)$ to the interval $]0,1[$, which is not always a desired property.

We can also note that $\sigma(z)$ is a shifted version of the tanh function:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1-e^{-2z}}{1+e^{-2z}} = 2\sigma(2z) - 1 \tag{5}$$

Hence the expression that shifts $\tanh(z)$ outputs towards the interval $]0,1[$ instead of $[-1,+1]$:

$$\sigma(z) = \frac{1}{2} \times \tanh\left(\frac{z}{2}\right) + \frac{1}{2} \tag{6}$$

2.1. Disadvantages of the Sigmoid Activation Function. The major disadvantage of the sigmoid function is its negative impact on the learning process, known in the literature as the problem of ”vanishing gradients” [13]. For many years this problem has constituted a serious obstacle to efficiently training deep neural networks, and the contribution of the sigmoid function to this problem was either misunderstood or underestimated. In fact, the problem results

from the application of the learning rule (2) to adapt the weights and biases of the network during the backward step of each learning epoch.

Concretely, the partial derivatives of E with respect to the weights of each unit i in each layer $l \leq L$ are computed using the chain rule. This results in a product of many terms, several of which are small terms in the range $]0, 0.25]$ for each partial derivative $\frac{\partial E}{\partial w_{ij}^{[l]}}$, $1 \leq j \leq n_l$. Consequently, as we progress towards the input layer, the computed derivatives and the corresponding corrective terms, $-\alpha \times \frac{\partial E}{\partial w_{ij}^{[l]}}$, become smaller and smaller, which undermines the convergence of the algorithm in reasonable amounts of time.

This is similar to the also well-known problem of *exploding gradients* observed with linear activation functions when the outputs increase exponentially with the number of layers, making the training of the network very difficult to achieve [14].

Moreover, as the input of each processing layer is the output of its previous layer, using the sigmoid activation function for all processing layers will restrict all outputs to the range $]0, 1[$ during the forward step. For the hidden layers, this is a hardly justifiable restriction because the primary goal of these layers is to learn a hierarchy of internal representations of the input data, not probabilities between the values 0 and 1.

2.2. Activation Functions for the Hidden Layers. Since the publication of the seminal paper, which introduced the rectified linear unit activation function (ReLU) [15], the sigmoid function is no longer used for hidden layers. In fact, ReLU and its numerous variants have proven to be more appropriate for hidden layers because they don't suffer from the problem of saturating units [16], which is a main cause of the vanishing gradients phenomenon. The original ReLU function is defined by the expression:

$$\text{ReLU}(z) = \max(0, z) \tag{7}$$

Its graph is depicted by Figure 4, along with the graph of its derivative dReLU.

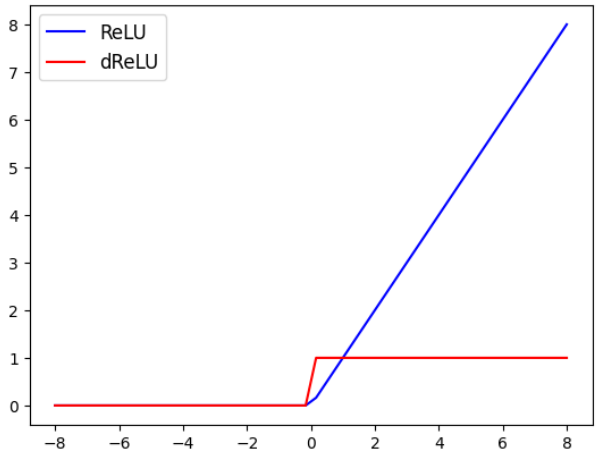

FIGURE 4. The original ReLU activation function and its derivative dReLU

The rectification introduced by ReLU to the linear activation function, $a(z) = z$, consists simply in replacing all negative values of $a(z)$ with 0. This ensures the non-linearity condition while avoiding the constraints of limiting the neuron outputs to values less than 1 and their derivatives to values less than 0.25.

However, despite the improvements achieved by the earlier adoption of ReLU, some concerns have rapidly emerged. One of them is the problem of *dead units*. These are units for which the linear part of their computing process produces a negative value for the dot product $z^{[l]} = \mathbf{w}^{[l]T} \cdot \mathbf{a}^{[l-1]}$. They are called "dead units" because they are not activated ($a(z^{[l]}) = 0$), and cannot therefore actively participate in the ongoing learning process.

To remedy this problem, several variants of ReLU have been proposed in the literature. A good review of these variants, as well as some alternatives to ReLU, can be found in reference. Among all these variants, the most famous and widely used are the *Leaky ReLU* (LReLU)5 and the *Parametric ReLU* (PReLU), defined by:

$$\text{LReLU}(z) = \begin{cases} z & \text{if } z \geq 0 \\ \alpha z & \text{if } z < 0 \end{cases} \tag{8}$$

where α is a user-defined parameter for LReLU, typically set to small values such as 0.01, and a trainable parameter for PReLU.

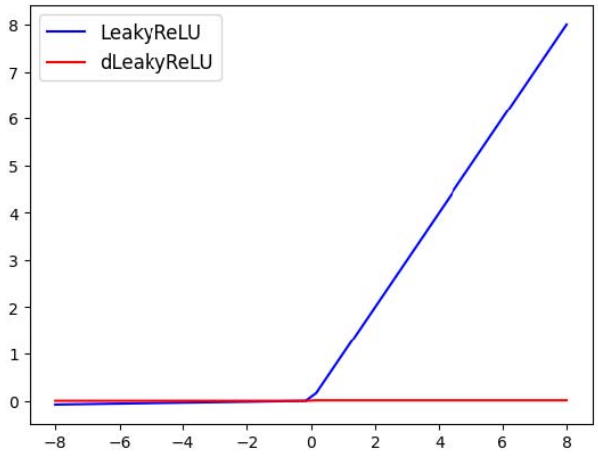

FIGURE 5. The Leaky ReLU activation function and its derivative dReLU

More recently, other activation functions have been proposed as alternatives to ReLU. Typical examples of this category of functions include the *Sigmoid-weighted Linear Unit (SiLU)*, which was proposed in 2017 for reinforcement learning[17], and its generalization, *Swish*, which was also proposed in the same year.

The SiLU 6 function is defined by:

$$\text{SiLU}(z) = z \cdot \frac{1}{1+e^{-z}} \tag{9}$$

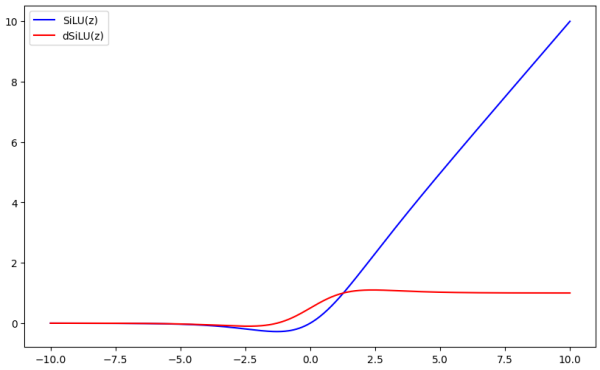

FIGURE 6. The SiLU activation function and its derivative dSiLU

which combines both linear and sigmoid behaviors. Its generalization, Swish, is a parametric version defined by:

$$\text{Swish}(z) = z \cdot \frac{1}{1+e^{-\beta z}} \tag{10}$$

where $\beta \in [0, \infty[$ is either a user-defined constant or a trainable parameter. From (10), we can easily see that for $\beta = 0$, the Swish function reduces to the linear function $f(z) = z$, and for $\beta \to \infty$, it approaches the ReLU function.

In practice, Swish has proven to be more effective than ReLU for deep architectures with more than 40 hidden layers. It is therefore the recommended choice for very deep models.

2.3. Binary Classification Case. Binary classification is the task of learning the best possible separation between training observations that originate from two different classes: a positive class, commonly designated by the label 1, and a negative class whose label is 0. When the two classes are not linearly separable, this task can be performed using a deep neural network with a single sigmoid neuron in the output layer.

The sigmoid is indeed the most suitable activation function for this neuron because it can be interpreted as a measure of the probability for the input object to belong to the positive class, i.e.,

$$p(y = 1 \mid \mathbf{x}; \mathbf{w}) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}} \tag{11}$$

In contrast, for the hidden layers, the sigmoid should be avoided and replaced by ReLU or one of its variants or alternatives discussed in the previous subsection.

2.4. Multi-class Classification Case. For multi-class classification problems, the output layer L of the network should contain c processing units, where c denotes the total number of classes in the training dataset. The most appropriate activation function for these units is the *softmax* function, defined for each unit i of the output layer L (Fig. 1) by the expression:

$$a_i^{[L]} = \frac{e^{Z_i^{[L]}}}{\sum_{k=1}^{c} e^{Z_k^{[L]}}} \tag{12}$$

where

$$Z_i^{[L]} = \mathbf{w}_i^{[L]T} \cdot \mathbf{a}^{[L-1]} \tag{13}$$

The advantage of the softmax activation function is that it produces a vector of c components that satisfy two important conditions: (1) $a_i^{[L]} \in]0, 1[$ for all $1 \leq i \leq c$, and (2) $\sum_{k=1}^{c} a_k^{[L]} = 1$.

Therefore, these components can be interpreted as a probability distribution of the input data over the c possible classes. They serve as a basis for assigning each object to a particular class.

The decision rule applied is the *maximum likelihood rule*, which assigns each object to the class for which the output neuron yields the highest probability. Hence the name *softmax*.

2.5. Case of Regression Problems. In regression problems, the computed outputs are continuous real values. Applying deep neural networks to these problems requires an output layer of c neurons, where c is the total number of target values to predict from the input data. The role of the training phase is to find the best possible estimation of the hypothetical function that maps input vectors to the corresponding outputs in the training set.

The number and size of the hidden layers can be heuristically chosen, similarly to classification problems. The recommended activation functions are ReLU (or one of its variants or alternatives) for the hidden layers, and a linear activation function for the output layer.

3. Experimental Results and Discussion

In this section we provide and discuss some experimental results that illustrate the importance of correctly choosing the activation functions, and the impact that an inappropriate function can have on the overall performance of a deep model regardless of all its remaining hyperparameters, including the depth. For this, we consider the well-known and publicly available MNIST handwritten digit dataset [16] and we propose to analyze it using 5 different models, starting with a simple model with one hidden layer, and progressively augmenting the number of hidden layers.

The MNIST dataset is composed of 70000 images of handwritten digits of size 28x28, 60000 of which are reserved for the training phase and the remaining 10000 for the test of generalization or model evaluation phase.

The design and implementation of each model, as well as its training and evaluation phases, are performed using the TensorFlow platform. Moreover, during all our experiments the following hyperparameters were set for the 5 models as follows: 1) loss function: cross-entropy, 2) optimizer: adam, which is a variant of the stochastic gradient descent [17], 3) performance evaluation: accuracy, or percentage of well-classified samples for both the training and the test

databases, 4) mini-batch size: 32. In stochastic descent gradient the mini-batch size represents the number of samples that should be analyzed before each adjustment of the trainable parameters at each epoch of the training process, and 5) epochs = 10, which is the number of training iterations to perform over all of the training data.

Model	**Hidden layers**			**Total number of trainable parameters**	**Training time (in sec)**	**Accuracy Training data (%)**	**Accuracy Test data (%)**
	nbr	**size**	**AF**				
1	1	128	relu	101707	81.97	94	95
2	2	128	relu	118282	41.15	96	96
3	5	128	sigmoid	184330	82.52	91	91
4	10	128	sigmoid	266890	81.99	80	80
5	15	128	sigmoid	348160	93.83	10	10

TABLE 1. Results Summary for the 5 Studied Models

The obtained numerical results are summarized in III. For each of the five studied models, the three first corresponding columns indicate the number of hidden layers, their size in number of neurons, and their activation function; while the remaining four columns are reserved for the numerical results of the model. A brief analysis of this table shows that the first two models, which are shallow networks with only one and two hidden layers of 128 relu units each, have achieved an accuracy of respectively 94% and 96% on the training data and 95% and 96% on the test data.

The remaining three models have deeper architectures with respectively 5, 10 and 15 hidden layers composed each with 128 sigmoid units. The reported results show clearly that the two shallow models performed better than the three deeper ones for both the training and the test data. This loss of performance is due to the problem of vanishing gradients caused by the sigmoid activation function used by the hidden layers. We can also note the decrease of accuracy with the number of hidden layers, which is also due to the vanishing gradients problem.

4. CONCLUSION

In this paper we were interested in the impact that activation functions can have on the global performance of deep neural networks. In particular, we addressed the close relationship between the vanishing gradients problem and the activation functions used by the hidden layers of the network. For many years after the publication of the back-propagation algorithm, which has proven to be successful in overcoming the limitations of perceptrons, researchers have been satisfied by the first activation function proposed in this algorithm, namely the sigmoid function, or its shifted version, tanh; but its role in the vanishing gradients problem has been neglected. With the publication of the ReLU function in 2012 a great raise in the awareness of the problem has been, however, observed, leading to many research studies in this field and to the proposition of many new activation functions.

The paper provides a historical review of the main contributions recently proposed in the literature, and shows that these contributions can be grouped in two main categories: i) new variants of ReLU, and ii) new alternatives to ReLU. It then gives some practical recommendations for guiding the choice of the most appropriate function for each layer of the network depending on the nature of each application. To show the extent to which an inappropriate choice of the activation function can impact the performance of a deep neural network, some experimental results are also presented and discussed. These results are obtained for the MNIST data using 5 distinct models with 5 different numbers of hidden layers ranging from 1 to 15. The main conclusion is that a shallow network, with only one or two hidden layers, can perform better than a deeper model with more than 10 layers only because its activation functions are more appropriately selected.

This encourages the continuation of this research topic, and one of the ideas that deserves further development in this framework is the automatic determination of parameters in the case of parametric activation functions. The same idea can also applied to other hyper-parameters, including weight initialization and regularization techniques.

CONFLICT OF INTERESTS

The authors declare that there is no conflict of interests.

REFERENCES

[1] A. El Bakali Kassimi, M. Madiafi, A. Kammour, A. Bouroumi, A Deep Neural Network for Detecting the Severity Level of Diabetic Retinopathy from Retinography Images, in: 2022 2nd International Conference on Innovative Research in Applied Science, Engineering and Technology (IRASET), IEEE, 2022, pp. 1-7. https://doi.org/10.1109/iraset52964.2022.9738202.

[2] M. Yayla, M. Günzel, B. Ramosaj, J. Chen, Universal Approximation Theorems of Fully Connected Binarized Neural Networks, arXiv:2102.02631 (2021). https://doi.org/10.48550/arXiv.2102.02631.

[3] Y. LeCun, Y. Bengio, G. Hinton, Deep Learning, Nature 521 (2015), 436–444. https://doi.org/10.1038/nature14539.

[4] S. Sun, Z. Cao, H. Zhu, J. Zhao, A Survey of Optimization Methods from a Machine Learning Perspective, IEEE Trans. Cybern. 50 (2020), 3668–3681. https://doi.org/10.1109/tcyb.2019.2950779.

[5] M. Hossin, M.N. Sulaiman, A Review on Evaluation Metrics for Data Classification Evaluations, Int. J. Data Min. Knowl. Manag. Process. 5 (2015), 01–11. https://doi.org/10.5121/ijdkp.2015.5201.

[6] R. Andonie, Hyperparameter Optimization in Learning Systems, J. Membr. Comput. 1 (2019), 279–291. https://doi.org/10.1007/s41965-019-00023-0.

[7] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet Classification with Deep Convolutional Neural Networks, Commun. ACM 60 (2017), 84–90. https://doi.org/10.1145/3065386.

[8] W. Mcculloch, W. Pitts, A Logical Calculus of the Ideas Immanent in Nervous Activity, Bull. Math. Biol. 52 (1990), 99–115. https://doi.org/10.1016/s0092-8240(05)80006-0.

[9] F. Rosenblatt, The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain., Psychol. Rev. 65 (1958), 386–408. https://doi.org/10.1037/h0042519.

[10] M. Minsky, S.A. Papert, Perceptrons, The MIT Press, 2017. https://doi.org/https://doi.org/10.7551/mitpress/11301.001.0001.

[11] D.E. Rumelhart, G.E. Hinton, R.J. Williams, Learning Representations by Back-Propagating Errors, Nature 323 (1986), 533–536. https://doi.org/10.1038/323533a0.

[12] H.H. Tan, K.H. Lim, Vanishing Gradient Mitigation with Deep Learning Neural Network Optimization, in: 2019 7th International Conference on Smart Computing & Communications (ICSCC), IEEE, 2019, pp. 1-4. https://doi.org/10.1109/icscc.2019.8843652.

[13] G. Philipp, D. Song, J.G. Carbonell, The Exploding Gradient Problem Demystified - Definition, Prevalence, Impact, Origin, Tradeoffs, and Solutions, arXiv:1712.05577 (2017). http://arxiv.org/abs/1712.05577v4.

[14] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet Classification with Deep Convolutional Neural Networks, in: Communications of the ACM, Association for Computing Machinery (ACM), 2017, pp. 84-90. https://doi.org/10.1145/3065386.

[15] P. Ramachandran, B. Zoph, Q.V. Le, Searching for Activation Functions, arXiv:1710.05941 (2017). http://arxiv.org/abs/1710.05941v2.

[16] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based Learning Applied to Document Recognition, in: Proceedings of the IEEE, Institute of Electrical and Electronics Engineers (IEEE), 1998, pp. 2278-2324. https://doi.org/10.1109/5.726791.

[17] D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization, arXiv:1412.6980 (2014). http://arxiv.org/abs/1412.6980v9.